Building a GNU GCC cross-compiler

NESTED SYSTEMS

Document revision

Author	Revision	Date	Comments
Patrick Beatini	1.0	09/23/12	Initial revision
Patrick Beatini	1.1	01/10/14	Add ARM compilation example.
Patrick Beatini	1.2	01/21/14	Format the document. Add qemu simulation for ARM.
Patrick Beatini	1.3	01/26/2014	Add newlib compilation with fno-short-enums.

Abstract

The objective of this white paper is to provide a *simple* procedure for building a GNU GCC cross compiler chain and the GDB debugger.

It also describes the build of the standard library called **newlib**. Finally, it provides a simple run-able example.

The gcc tools chain described in this document is the build of a cross-compiler for a PowerPC eabi (32 bits) and an ARM eabi from a ia64 LINUX host with the following characteristics:

- Hardware
  - Memory: 4Go
  - Processor Intel G630 dual core
- Operating system
  - LINUX Ubuntu
  - Release 10.04 (lucid)

The cross compiler is built with 5 steps:

- Step 1: Create shell variables to simplify the commands. They will be used during the process.
- Step 2: binutils build: These tools are required for the cross-compiler builds.
- Step 3: GCC bootstrap build: The final GCC cannot be built directly since the headers and the libraries should be in the final CPU format. This intermediary step creates a compiler tool chains needed for the newlib build.
- Step 4: Build the newlib in the target CPU format.
- Step 5: Build the final GCC in the target CPU format.

GDB is built with 3 steps.

- Step 1: Build the termcap library required by GBD.
- Step 2: Build the expat library required only if GDB is used remotely.
- Step 3: Build GDB.

Important notes

- The build process is ordered.
- For each source being compiled, it is important to create a build directory for each expanded tar-balls. **The build directory MUST be created from the source directory**.
- All installation commands (make install) should be done as **root**. For Ubuntu, these commands will be preceded by: **sudo**.
- If the target CPU accepts hard floating point, then remove the **--with-float=soft** and **--enable-soft-floats** switches from the configure commands.

1. Code sources

The different utilities built in this papers are submitted to the GPL license. The following list provides the code sources required as well as their location:

1. gcc-4.7.2: http://gcc.gnu.org/
2. binutils-2.22: http://www.gnu.org/software/binutils/:
3. newlib-1.20.0: ftp://sources.redhat.com/pub/newlib/index.html
4. gdb-7.5: http://www.gnu.org/software/gdb/download/
5. termcap-1.3.1: http://ftp.gnu.org/gnu/termcap/
6. expat-2.1.0: http://sourceforge.net/projects/expat/files/expat/2.1.0/
7. qemu-1.7.0: http://wiki.qemu.org/Download

Note
Ensure that the following GNU tools are present on your system:

- texinfo
- flex
- bison

2. Shell variables definition

To simplify the build process, the following Shell variables are defined:

- TARGET: Define the type of the compiler.
- PREFIX: Define the file name prefix to be added to the generated tools and libraries.

The usual PATH environment variable is modified to include the location of the generated binaries.

Notes

- The ARM compiler uses by default variable sizes for enumerated type. If a fixed size is expected, the GCC flags **-fno-short-enums** should be used while compiling the **newlib**. In this case, the enumerated size will be 4 bytes for 32 bits CPU (same size than int). To pass such flags during the GCC build, the **CFLAGS** environment variable should be defined with the appropriate setting:
  ```
  export CFLAGS=-fno-short-enums
  ```
  Using this setting requires that the platform compilation (built with the new ARM compiler) should also include this flag.
- **All configure and make commands must be entered under a shell defining these variables.**

3. Binutils build

The following steps generate the initial tools that will be used for the GCC compilation.

The build process is the following:

1. Uncompress the binutils source and move to its directory.
2. Create the build directory
3. Move to the build directory
4. Configure the build
5. Make and install

PowerPC	ARM
1. - 2. mkdir ppc-build 3. cd ppc-build 4. ../configure --target=$TARGET \ --prefix=$PREFIX --enable-soft-float 5. make all sudo make install	1. - 2. mkdir arm-build 3. cd arm-build 4. ../configure --target=$TARGET \ --prefix=$PREFIX --enable-interwork \ --enable-multilib 5. make all sudo make install

Notes

- The option --enable-soft-float indicates that the float are emulated by software.
- The option --enable-interwork indicates that the mix of Thumb and ARM instruction is allowed.
- The option --enable-multilib instructs the build process to generate several form of libraries (for example, big endian, little endian, thumb, and so on).

4. GCC bootstrap build

This first GCC build constructs a usable C compiler in the target format, which will be used to compile the newlib library.

The general process is the following:

1. Uncompress the gcc source and move to its directory.
2. Apply the prerequisites
3. Create the build directory
4. Move to the build directory
5. Configure the build
6. Make and install

PowerPC	ARM
1. - 2. ./contrib/download_prerequisites 3. mkdir ppc-build 4. cd ppc-build 5. ../configure --target=$TARGET \ --prefix=$PREFIX --with-newlib \ --without-headers --with-gnu-as --with-gnu-ld\ --disable-shared --disable-libssp \ --disable-libada --enable-soft-float \ --with-float=soft --enable-languages="c,c++" 6. make all sudo make install	1. - 2. ./contrib/download_prerequisites 3. mkdir arm-build 4. cd arm-build 5. ../configure --target=$TARGET \ --prefix=$PREFIX --enable-interwork \ --enable-multilib --with-newlib \ --without-headers --with-gnu-as \ --with-gnu-ld --disable-shared \ --disable-libssp --disable-libada \ --enable-languages="c,c++" 6. make all sudo make install

Note

The build configuration indicates that the compiler chain will use **newlib** but for now the headers are not included.

5. Newlib build

The following steps generate the **newlib** library in the target format.

The general process is the following:

1. Uncompress the newlib source and move to its directory.
2. Create the build directory
3. Move to the build directory
4. Create the RANLIB_FOR_TARGET environment variable
5. Configure the build
6. Make and install

PowerPC	ARM
1. - 2. mkdir ppc-build 3. cd ppc-build 4. export RANLIB_FOR_TARGET=\ /usr/local/powerpc-eabi/bin/powerpc-eabi-ranlib 5. ../configure --target=$TARGET \ --prefix=$PREFIX --with-gnu-as \ --with-gnu-ld --disable-shared --disable-libssp\ --disable-libada --enable-soft-float \ --with-float=soft 6. make all sudo make install	1. - 2. mkdir arm-build 3. cd arm-build 4. export RANLIB_FOR_TARGET=\ /usr/local/arm-none-eabi/bin/arm-none-eabi-ranlib 5. ../configure --target=$TARGET \ --prefix=$PREFIX --enable-interwork \ --enable-multilib --with-gnu-as \ --with-gnu-ld --disable-shared \ --disable-libssp --disable-libada \ --disable-newlib-supplied-syscalls \ --enable-lite-exit --disable-newlib-multithread 6. make all sudo make install

Notes

- If the RANLIB_FOR_TARGET variable is not defined as above, the following error occurs:
  ```
  ( cd '/usr/local/powerpc-eabi/powerpc-eabi/lib' && powerpc-eabi-ranlib libm.a )
  /bin/bash: line 5: powerpc-eabi-ranlib: command not found
  make[4]: *** [install-toollibLIBRARIES] Error 127
  ```
- The ARM configuration instructs the build to disable the system calls via the option **disable-newlib-supplied-syscalls** and to use the light exit format. This is required for targets for which the system calls are not implemented via the **swi** assembler instruction. The power-PC does not require these options.
- If the target operating system is not multi-threaded or if it is non preemptive multi-threaded, then it could be convenient to disable the reentrant mechanism of **newlib**. This is accomplished by setting the option flag **disable-newlib-multithread.**

6. Final GCC build

The following steps generate the final **gcc** tools chain with C and C++ languages.

If you wish to use the ARM tools chain to create an executable that can be simulated with **qemu**, you should modify the gcc configuration as follow before starting the build:
File: gcc/config/arm/unknown-elf.h:
Replace:
```
#define UNKNOWN_ELF_STARTFILE_SPEC   " crti%O%s crtbegin%O%s crt0%O%s"
```
by
```
#define UNKNOWN_ELF_STARTFILE_SPEC   " crt0%O%s crti%O%s crtbegin%O%s"
```

The general process is the following:
1. Move to the gcc directory
2. Move to the build directory
3. Remove the previous compilation result
4. Configure the build
5. Make and install

PowerPC	ARM
1. - 2. cd ppc-build 3. rm -rf * 4. ../configure --target=$TARGET \ --prefix=$PREFIX --with-newlib \ --with-gnu-as --with-gnu-ld --disable-shared \ --enable-soft-float --with-float=soft \ --enable-languages="c,c++" 5. make all sudo make install	1. - 2. cd arm-build 3. rm -rf * 4. ../configure --target=$TARGET \ --prefix=$PREFIX --with-newlib \ --with-gnu-as --with-gnu-ld \ --enable-interwork --enable-multilib \ --disable-shared --enable-languages="c,c++" 5. make all sudo make install

Notes
- This build takes into account the newlib headers (option --without-headers not more present).
- Modification of the *gcc/config/arm/unknown-elf.h* file:
  - i. This specification file is used **only** when gcc is built for a arm-none-eabi target. So, its modification does not affect other gcc tools chain.
  - ii. The modification consists of reordering the file list defined by the macro: UNKNOWN_ELF_STARTFILE_SPEC. This macro is used by the linker to respect the order of the start files insertion in the executable (executable here means the code we are building for our target and not the gcc compilation).
  - iii. The file crt0 (generated by the newlib build) contains the _start function, which initializes the libraries. The files crtbegin and crti contain the table of global constructors/desctructors. The crtbegin initialization is called by the _start function of crt0.
  - iv. Usually, the order of these start files must be crt0 first, then crtbegin and crti.
  - v. When qemu is used, it does not pay attention to the _start entry point. For its point of view, the code starts at 0x10000. So, a target executable code built without the gcc correction will fail a qemu simulation since the crtbegin code will be before the _start code.

7. Termcap library build

The following steps generate the **temcap** library required by GDB.

The general process is the following:
1. Uncompress the termcap source and move to its directory.
2. Create the build directory
3. Move to the build directory
4. Configure the build
5. Make and install

PowerPC	ARM
1. -	1. -
2. mkdir ppc-build	2. mkdir arm-build
3. cd ppc-build	3. cd arm-build
4. ../configure --target=$TARGET \ --prefix=$PREFIX	4. ../configure --target=$TARGET \ --prefix=$PREFIX
5. make all sudo make install	5. make all sudo make install

8. Expat library build

The following steps generate the **expat** library needed by GDB if a remote debugging is expected.

The general process is the following:

1. Uncompress the expat source and move to its directory.
2. Create the build directory
3. Move to the build directory
4. Configure the build
5. Make and install

PowerPC	ARM
1. -	1. -
2. mkdir ppc-build	2. mkdir arm-build
3. cd ppc-build	3. cd arm-build
4. ../configure --target=$TARGET \ --prefix=$PREFIX	4. ../configure --target=$TARGET \ --prefix=$PREFIX
5. make all sudo make install	5. make all sudo make install

9. GDB build

The following steps generate GDB. In this build, we expect GDB run-able in simulation
GDB has a code simulation for the PowerPC. However this feature is not available for the ARM processor.
So, the options chosen for the PowerPC GDB is the code simulation. The option chosen for ARM is the remote debugging (usable with qemu).

The general build process is the following:

1. Uncompress the gdb source and move to its directory.
2. Create the build directory
3. Move to the build directory
4. Create the environment variable locating the termcap and the expat libraries.
5. Configure the build
6. Make and install

PowerPC	ARM
1. - 2. mkdir ppc-build 3. cd ppc-build 4. export LDFLAGS=-L$PREFIX/lib 5. ../configure --target=$TARGET \ --prefix=$PREFIX --enable-sim-powerpc \ -- enable-sim-stdio 6. make all sudo make install	1. - 2. mkdir arm-build 3. cd arm-build 4. export LDFLAGS=-L$PREFIX/lib 5. ../configure --target=$TARGET \ --prefix=$PREFIX --with-expat 6. make all sudo make install

10. PowerPc tests

In this section, we check our new PowerPC build chain against a simple example. Then, the example is simulated under GDB.

Create the following hello.c file

```
# include <stdio.h>

int main()
{
    char buffer[256];

    printf("Hello world\n");
    return 0;
}
```

Build it in the target format

```
/usr/local/powerpc-eabi/bin/powerpc-eabi-gcc -mcpu=405 hello.c -o hello -msim
```

Dump the executable (elf format)

```
/usr/local/powerpc-eabi/bin/powerpc-eabi-objdump -f hello
hello:     file format elf32-powerpc
architecture: powerpc:common, flags 0x00000112: EXEC_P, HAS_SYMS, D_PAGED
start address 0x10000098
```

Execute it via GDB

```
/usr/local/powerpc-eabi/bin/powerpc-eabi-run hello
Hello world
```

Notes

- To compile with specific memory map, use the -T option as follow: /usr/local/powerpc-eabi/bin/powerpc-eabi-gcc -Wl,-Ttext,0x4000,-Tdata,0xf000 hello.c -msim (-Wl,-Ttext,0x4000,-Tdata,0x10000)
- The **-msim** option indicates that the code is run'able with GDB target code simulation.

11. ARM tests

In this section, we check our new ARM build chain against a simple example. Then, the example is simulated under qemu. The test is more complex than the previous one. This is due to the fact that qemu simulate a real embedded platform.
The platform chosen is the **versatile-pb**.

Create the following hello.c file

```
# include <stdio.h>
volatile unsigned char * const UART0_PTR = (unsigned char *)0x101f1000;

void display(const char *string)
{
    while(*string != '\0'){
        *UART0_PTR = *string;
        string++;
    }
}

void _exit(int code)
{
    while (1) {
    }
}

unsigned char* _sbrk(unsigned int size)
{
    return NULL;
}

int main()
{
    char buffer[256];

    display("Hello World\n");

    sprintf(buffer, "Value: 0x%x\n", 0x1234);
    display(buffer);
}
```

Create the following linker file

```
RAM_BASE_ADDR = 0; /* RAM start address */
RAM_START_ADDR = 0x10000; /* Start address for the code */
RAM_SIZE = 0x01000000; /* RAM size: 64 Mo */

SECTIONS
{
    . = RAM_START_ADDR;
    .text ALIGN(4) : {*(.text)}

    .init ALIGN(4) : {*(.init)}

    .fini ALIGN(4) : {*(.fini) }
    PROVIDE(__etext = .);
    PROVIDE(_etext = .);
    PROVIDE(etext = .);

    .rodata ALIGN(4) : {
      *(.rodata)
      *(.rodata.str1.4)
      *(.rodata.cst4)
      *(.rodata1)
    }

    .gcc_except_table : {*(.gcc_except_table)}
    .ctors ALIGN(4) : {
       PROVIDE(__CTOR_COUNT__ = .);
       LONG((__CTOR_END__ - __CTOR_LIST__) / 4 -2)  /* number of constructor */
       PROVIDE(__CTOR_LIST__ = .);
        KEEP(*(SORT(.ctors)))
        LONG(0);
        PROVIDE(__CTOR_END__ = .);
    }

    .dtors ALIGN(4) : {
       PROVIDE(__DTOR_COUNT__ = .);
       LONG((__DTOR_END__ - __DTOR_LIST__) / 4 -2)  /* number of destructor */
       PROVIDE(__DTOR_LIST__ = .);
        KEEP(*(SORT(.dtors)))
        LONG(0);
        PROVIDE(__DTOR_END__ = .);
    }
    .got : {
       *(.got.plt)
       *(.got)
     }
    .got2 : { *(.got2)}


    .= ALIGN(0x4);
    PROVIDE(__sdata = .);
    .data ALIGN(4) : {
       *(.sdata2)
        . = ALIGN (0x8);
       *(.data)
       *(.data.rel.local)
       *(.data.rel.ro.local)
       *(.data.rel)
       *(.sdata)
    }
     PROVIDE(__edata = .);
     PROVIDE(_edata = .);
     PROVIDE(edata = .);

    .bss ALIGN(4) (NOLOAD) : {
      PROVIDE(__sbss_start = .);
      *(.sbss)
      *(.sbss2)
      PROVIDE(__sbss_end = .);
      PROVIDE(__sbss_end__ = .);
      PROVIDE(__bss_start = .);
      PROVIDE(__bss_start__ = .);
      *(.bss)
      PROVIDE(__bss_end = .);
      PROVIDE(__bss_end__ = .);
      *(COMMON)
      _end = .;
      PROVIDE(_end = .);
      PROVIDE(end = .);
      PROVIDE(__end__ = .);
    }
    PROVIDE(__stack = RAM_BASE_ADDR  + RAM_SIZE - 128);
    PROVIDE(_stack = RAM_BASE_ADDR  + RAM_SIZE - 128);
}
```

Compile hello.c

/usr/local/arm-none-eabi/bin/arm-none-eabi-gcc -c -g -mcpu=arm926ej-s hello.c -o hello.o

Link and generate the hello.elf executable

```
/usr/local/arm-none-eabi/bin/arm-none-eabi-gcc -o hello.elf hello.o -L/usr/local/arm-none-eabi/lib -T linker.ld\
 -Xlinker --no-enum-size-warning -Xlinker --print-map > hello.map
```

Make the elf executable as a binary one (for qemu)

```
/usr/local/arm-none-eabi/bin/arm-none-eabi-objcopy -O binary hello.elf hello
```

Simulate the code (without GDB)

```
qemu-system-arm -M versatilepb -m 64 -nographic -kernel hello
Hello World
Value: 0x1234
```

Stop the simulation (ctrl + a, then x)

```
QEMU: Terminated
```

Simulate the code with qemu and control it with GDB

Start qemu and freeze it

```
   qemu-system-arm -S -s -M versatilepb -m 64 -nographic -kernel hello
```

Open another terminal and move to the test directory

Start GDB

```
   /usr/local/arm-none-eabi/bin/arm-none-eabi-gdb
```

Under GDB, connect to the target, load the symbol and start the executable

```
  target remote localhost:1234
  symbol-file hello.elf
  continue
```

Note 1

The linker flag **-Xlinker –no-enum-size-warning** is needed to avoid the warnings:

```
/usr/local/arm-none-eabi/lib/gcc/arm-none-eabi/4.7.2/../../../../arm-none-eabi/bin/ld:
warning: hello.o uses variable-size enums yet the output is to use 32-bit enums; use of
enum values across objects may fail
/usr/local/arm-none-eabi/lib/gcc/arm-none-eabi/4.7.2/../../../../arm-none-eabi/bin/ld:
warning: /usr/local/arm-none-eabi/lib/gcc/arm-none-eabi/4.7.2/../../../../arm-none-
eabi/lib/libc.a(lib_a-svfprintf.o) uses variable-size enums yet the output is to use 32-
bit enums; use of enum values across objects may fail
/usr/local/arm-none-eabi/lib/gcc/arm-none-eabi/4.7.2/../../../../arm-none-eabi/bin/ld:
warning: /usr/local/arm-none-eabi/lib/gcc/arm-none-eabi/4.7.2/../../../../arm-none-
eabi/lib/libc.a(lib_a-svfiprintf.o) uses variable-size enums yet the output is to use
32-bit enums; use of enum values across objects may fail
```

These warnings indicate that the files **lib_a-svfprintf.o, lib_a-mbtowc_r.o** and **lib_a-svfiprintf.o** have been generated without the -fno-short-enums compile flags.
In fact, the newlib makefile forces these files to be explicitly compiled with the flag -fshort-enums, which overrides our setting.
The object files **lib_a-svfprintf.o** and **lib_a-svfiprintf.o** are generated from vfprintf.c.

Note 2

Since qemu emulates a real embedded platform without any operating system, some basic functions should be provided:
File hello.c content:

- **display**: The standard I/O (console) are emulated via the UART0. The purpose of the **display** function is the sending of the displayed characters via the UART. The UART0 is located at the physical address 0x101f1000 for this platform.
- **sbrk**: The standard library requires a basic memory allocation function. The **_srbk** function is defined for this purpose. Since we do not use the memory allocation, this function returns NULL.
- **_exit**: The standard library requires a termination function. This function must not return.